


Spiking Neural P System Simulations on a High Performance GPU Platform

View metadata, citation and similar papers at core.ac.uk

brought to you by  CORE

provided by idUS. Depósito de Investigación Universidad de Sevilla

Miguel A. Martínez-del-Amor², and Mario J. Pérez-Jiménez²

¹ Algorithms & Complexity Lab, Department of Computer Science,
University of the Philippines Diliman
Diliman 1101 Quezon City, Philippines
fccabarle@up.edu.ph, hmadorna@dcs.upd.edu.ph

² Research Group on Natural Computing, Department of Computer Science
and Artificial Intelligence, University of Seville,
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
{mdelamor,marper}@us.es

Abstract. In this paper we present our results in adapting a Spiking Neural P system (SNP system) simulator to a high performance graphics processing unit (GPU) platform. In particular, we extend our simulations to larger and more complex SNP systems using an NVIDIA Tesla C1060 GPU. The C1060 is manufactured for high performance computing and massively parallel computations, matching the maximally parallel nature of SNP systems. Using our GPU accelerated simulations we present speedups of around 200× for some SNP systems, compared to CPU only simulations.

Keywords: Membrane computing, Spiking Neural P systems, GPU computing, CUDA, parallel computing.

1 Introduction

P systems are by nature distributed, parallel, and non-deterministic computing models defined within *Membrane computing*, which is a research area initiated by Gheorghe Păun in 1998 [16]. The objective, as with other disciplines of *Natural computing* (e.g. DNA/molecular computing, quantum computing, etc.), is to obtain inspiration from the way nature computes to provide efficient solutions to the limitations of conventional models of computation e.g. a Turing machine. Membrane computing can be thought of as an extension of DNA or molecular computing, zooming out from the individual molecules of the DNA and including other parts and sections of living cells in the computation, introducing the concept of distributed computing as well [16].

P systems are *abstractions* of the compartmentalized structure and parallel processing of biochemical *information* in biological cells. There are several P system *variants* defined in literature, each one based on the abstraction of different aspects (or ingredients) from cells, and that many of them have been proven



to be *computationally complete* [5]. There are three general classifications of P systems considering the level of abstraction: *cell-like* (a rooted tree where the *skin* or outermost cell membrane is the root), *tissue-like* (a graph connecting the cell membranes) and *neural-like* (a directed graph, inspired by *neurons* interconnected by their axons and synapses). The last type refer to *Spiking Neural P systems* (in short, SNP systems), where the *time difference* (when neurons fire and/or spike) plays an essential role in the computations [11]. An interesting result of P systems is that they are able to solve computationally *hard problems* (e.g. NP-complete problems) usually in *polynomial*, often linear *time*, but usually requiring *exponential space* as trade off [16].

Due to the nature of P systems, they are yet to be fully implemented *in vivo*, *in vitro*, or even *in silico*. Thus, practical computations of P systems are driven by silicon-based simulators. There are several simulators for P systems implemented over different software and hardware technologies [7]. In practice, P system simulations are limited by the physical laws of silicon architectures, which are often inefficient or not suitable when dealing with P system features, such as massive parallelism. However, in order to improve the efficiency of the simulators, it is necessary to exploit current technologies, leading to solutions in the area of *High Performance Computing (HPC)*, such as *accelerators* or many-core processors. In this respect, *Graphics Processing Units (GPUs)* have been consolidated as accelerators thanks to their throughput-oriented and highly-parallel architecture [9].

Several simulators for P systems have been developed over highly parallel platforms, including reconfigurable hardware as in FPGAs [14], CPU-based clusters [6], as well as in NVIDIA corporation's *Compute Unified Device Architecture (CUDA)* enabled GPUs [4,3]. These efforts show that parallel devices are very suitable in accelerating the simulation of P systems, at least for *transition* and *active membrane* P systems [3,4]. Efficiently simulating a Spiking Neural P (SNP) system, the P system variant of interest in this work, would thus require new efforts in parallel computing. Since SNP systems have already been represented as matrices due to their graph-like properties [18], simulating them in parallel devices such as GPUs is the next natural step. Matrix algorithms are well known in parallel computing literature, including GPUs [8], due to the highly parallelizable nature of linear algebra computations mapping directly to the data-parallel GPU architecture.

An SNP system simulator using CUDA was presented in [1] and [2]. These previous works however were executed in GPUs of workstations only, hence we intend to do better. We adapt and analyse the performance of this simulator on a high-end GPU NVIDIA Tesla C1060, designed ground-up for parallel computing and HPC, by simulating SNP systems of different sizes. A final simulator for SNP systems using CUDA would allow the designers to check their models, and perform other complex computations such as computing backwards.

This paper is organized as follows: Section 2 and 3 provide backgrounds for CUDA and SNP systems, respectively. The design of the simulator and simulation results are given in Section 4 and 5, respectively.

2 GPU Computing and NVIDIA CUDA

As many-core based platforms, GPUs are massively data-parallel processors which have high *chip scalability* in terms of processing units (cores, threads), and high bandwidth with internal GPU memories. The architectural difference between CPUs and GPUs is the reason why the latter offer larger performance increase over CPU only implementation of parallel code working on large amounts of input data [12]. The main *advantages* of using GPUs are their *low-cost*, *low-maintenance* and *low power consumption* relative to conventional parallel clusters and setups, while providing comparable or improved computational power [10]. For example, the latest GPUs of NVIDIA with 512 cores are readily available at consumer electronics stores for around \$500. GPUs can be programmed using a framework introduced by NVIDIA in 2007 called CUDA [12]. CUDA is a programming model and hardware architecture for general purpose computations in NVIDIA's GPUs [12]. The programmer can use CUDA for free of charge (including the compiler, driver, SDK, libraries, etc), and is easy to learn because it's an extension of the *C* language.

CUDA implements a *heterogeneous computing* architecture, where two different parts are often considered: the *host* (CPU side) and the *device* (GPU side). The host part of the code is responsible for controlling the program execution flow, transferring data to and from the device memory, and executing specific codes, called *kernel* functions, on the device. The device acts as a parallel *co-processor* to the host. The host *outsources* the parallel part of the program as well as the data to the device, since it is more suited to parallel computations than the host. The kernel code is executed in the device by a set of threads. They are organized into a three-level hierarchy, from highest to lowest: a grid of thread blocks, blocks of threads, and threads which can share data through shared memory and can perform simple barrier synchronization [12,15]. Using *kernel functions*, the programmer can specify the GPU resources: up to 65,535 blocks and up to 512 threads per block.

3 Spiking Neural P Systems

Now we first formally define SNP systems as computing models. An SNP system without delay, of degree $m \geq 1$, is of the form

$$\Pi = (O, \sigma_1, \dots, \sigma_m, syn, in, out),$$

where:

[1.] $O = \{a\}$ is the alphabet made up of only one object a , called spike; [2.] $\sigma_1, \dots, \sigma_m$ are m number of neurons of the form

$$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$

where:

(a) $n_i \geq 0$ gives the initial number of spikes (a) contained in neuron σ_i ; (b) R_i is a finite set of rules of the following forms:

(b-1) $E/a^c \rightarrow a^p$, are *Spiking rules*, where E is a regular expression over a , $c \geq 1$, and $p \geq 1$ number of spikes are produced (with the restriction $c \geq p$), transmitted to each adjacent neuron with σ_i as the originating neuron, and $a^c \in L(E)$; $a^k \rightarrow a^p$, is a special case of (b-1) where $L(E) = \{a^c\}$, $k = c, p = 1$; (b-2) $a^s \rightarrow \lambda$, are *Forgetting rules*, for $s \geq 1$, such that for each rule $E/a^c \rightarrow a^p$ of type (b-1) from R_i , $a^s \notin L(E)$; [3.] $syn = \{(i, j) \mid 1 \leq i, j \leq m, i \neq j\}$ are the synapses i.e. connection between neurons; [4.] $in, out \in \{1, 2, \dots, m\}$ are the input and output neurons, respectively.

The system works as follows: At any given time, a σ_i (neuron) should use *exactly* one rule only, if and only if the condition $a^c \in L(E)$ is met. This condition means as long as the multiplicity of spikes is in the language generated by the regular expression E , a rule (or several of them) is (are) applicable. The rule to be used or applied is chosen non-deterministically. If a *spiking rule* is used, after rule application c spikes are consumed in the σ_i , producing p number of spikes to all other σ_j such that $(i, j) \in syn$. If a *Forgetting rule* is applied, s number of a are removed from σ_i and no a or spike is produced. A global clock is followed by the system. Parallelism is at the system level, although each neuron works sequentially.

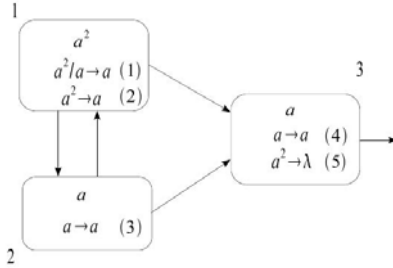


Fig. 1. Π_1 generates the set $\mathbb{N} - \{1\}$. Π_1 outputs are the time differences between the *first spike* of σ_3 and its succeeding spikes. A total ordering of the neurons is seen (σ_1 to σ_3) including a total ordering of the rules (1 to 5).

We designate the SNP system shown in Figure 1 as Π_1 [18]. For our simulations we use 2 additional systems: Figure 8 in [11] and Figure 14 in [11] which we designate as Π_2 and Π_3 respectively.

Next we present the matrix representation of an SNP system and its computations. This representation makes use of the following vectors and matrix definitions:

Configuration vector. C_k is the vector containing all spikes in every σ on the k th computation step/time. C_0 is the initial C_k of the system.

Spiking vector. S_k shows, at a given C_k , if a rule is applicable (having value 1) or not (having value 0 instead).

Spiking transition matrix. M_{SNP} is a matrix comprised of a_{ij} elements where a_{ij} is given as: $-c$ if rule r_i is in σ_j and is applied consuming c spikes; p if rule

r_i is in σ_s ($s \neq j$ and $(s, j) \in syn$ and is applied producing p spikes in total; 0 if rule r_i is in σ_s ($s \neq j$ and $(s, j) \notin syn$). The spiking transition matrix M_{Π_1} is shown in equation (1).

$$M_{\Pi_1} = \begin{pmatrix} -1 & 1 & 1 \\ -2 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & -2 \end{pmatrix} \quad (1)$$

Equation (2) provides the configuration vector at the $(k + 1)th$ step:

$$C_{k+1} = C_k + S_k \cdot M_{\Pi} \quad (2)$$

For Π_1 $C_0 = \langle 2, 1, 1 \rangle$. and we have the $S_0 = \langle 1, 0, 1, 1, 0 \rangle$ given its C_0 . Note that a second alternative $S'_0 = \langle 0, 1, 1, 1, 0 \rangle$, is possible if we use rule (2) over rule (1) instead (but not both at the same time). *Validity* in this case means that only one among several applicable rules is used and thus represented in the S_k . The C_0, S_0 for Π_2 and Π_3 can be similarly shown.

4 Parallel SNP System Simulation on GPU

We designate the improved SNP system simulator in this paper as *snpgpu-sim4* which is an update to *snpgpu-sim3* produced in [2]. Among the improvements of *snpgpu-sim4* over *snpgpu-sim3* include the use of multiple thread-blocks to accommodate matrices more than 512 elements, and a more streamlined part of the simulation code for handling the relationships between R_i , C_k , and S_k . This section will further expound on these, among other things.

The simulator takes in 3 inputs: *Mf*, *C0f*, and *Rf* which are the file counterparts of M , C_0 , and R_i , respectively. *Skf* is the file counterpart of S_k , which is produced by the simulator itself once it is run. *PyCUDA* was used in addition to conventional Python and CUDA C languages. PyCUDA is a Python wrapper for NVIDIA CUDA C and C++, enabling programmers to create GPU software using Python, and has been used for high performance computing [13]. The inputs are text files with delimiters, between rule to another rule in a σ and between σ s themselves. The elements of M are entered in *row-major order format* into the file, and are mapped onto each thread of a thread block, within the block grid as shown in Figure 3.

Figure 2 shows an instance of host-device interaction. The host functions sequentially and calls the kernel function/s. The device is split up into a grid of thread blocks, each with their own threads, and operate on the data in a *single program, multiple data* (SPMD) programming style [12]. The simulation algorithm is shown in Algorithm 1, which also indicates where a specific part of the simulation runs on (either host or device parts). Part I loads the 3 initial inputs and the succeeding inputs from their file counterparts, checking for formatting and pre-processing them for Part II. Part II, from Part I's outputs and from

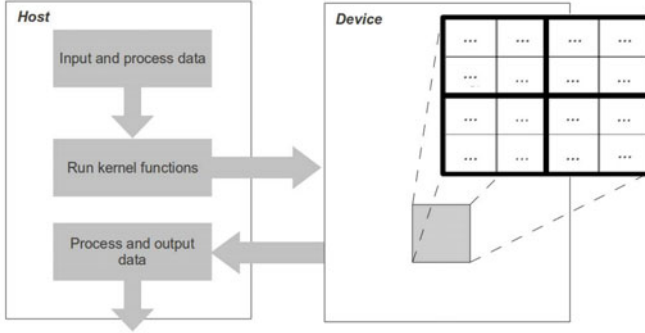


Fig. 2. Diagram showing a single run of the simulation flow. The host runs sequentially while the device is made up of a grid of thread blocks, each with their own threads operating in parallel.

Require: Input files: Ck, M, r .

- I. **(HOST)** Load input files. Mf, Rf are loaded once only. $C0f$ is also loaded once, then $Ckfs$ afterwards.
- II. **(HOST)** Determine if a rule in Rf is applicable based on the number of spikes present in each σ seen in Ckf . Then generate all valid and possible spiking vectors in a list of lists Skf .
- III. **(DEVICE)** Run kernel function on all valid and possible $Skfs$ from the current Ckf . Produce the next file configuration counterparts of $C_k + 1s$ and their corresponding $Skfs$.
- IV. **(HOST+DEVICE)** Repeat steps I to IV, till at least one of the two *Stopping criteria* is encountered.

Algorithm 1. Overview of SNP system simulation algorithm

Ckf and Rf , produces all the valid and possible $Skfs$. Part II produces all valid and possible Skf files as follows: For each n_i of σ_i , the $\{1,0\}$ strings are produced on a per neuron level. For example, for Π_1 we have $n_1 = 2$ for σ_1 . Now we have σ_1 strings '10' (choose to use R_1 instead of R_2) and '01' (choose to use R_2 over R_1). We only have one string for σ_2 , the string '1', since σ_2 has only one rule and it is readily applicable. Neuron σ_3 produces only one string also, '10', since only one rule is applicable given its $n_3 = 1$. Only R_4 is used in σ_3 and not R_5 . Once all the neuron level $\{1,0\}$ strings are produced, the strings are exhaustively paired up with the other strings in the other σ s from left to right as the ordering is important. The output therefore of Part II in this example given $C_k = \langle 2, 1, 1 \rangle$ are $(1,0,1,1,0)$ and $(0,1,1,1,0)$. Elements of the input files are treated as *strings* up to this point, because of the the concatenation and regular expression checking processes, among others.

Part III now treats the input elements as integral values. Equation 2 is performed in parallel such that each thread is either adding or multiplying a (matrix or vector) element. Once the C_{k+1} are produced by the device, the results are

moved back to the host. Part IV then checks whether to proceed or to stop based on 2 stopping criteria for the simulation: (I) if a *zero vector* (vector of zeros) is encountered, (II) if the succeeding C_k s have all been produced in previous computations. Both (I) and (II) make sure that the simulation halts and does not enter an infinite loop.

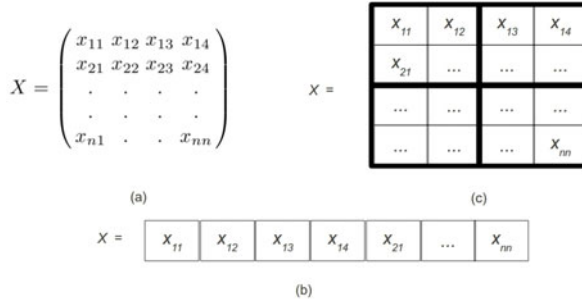


Fig. 3. Different representations of a given matrix X : (a) original matrix form (b) linear array in row-major order form, (c) using CUDA thread blocks in a single thread block grid. The linear array shows how the array’s elements are laid out: a (4×4) grid made up of (2×2) thread blocks. Each thread in a thread block computes a unique element of the array in parallel, and all of them execute the same kernel function.

5 Simulation Results and Observations

The simulations in this paper were executed using an Intel Xeon E5504 quad core CPU running at 2 GHz per core (there are two of these CPUs so there are effectively 8 cores). Each core has a 4MB cache. The GPU is an NVIDIA Tesla C1060 high performance GPU with 240 streaming-processor (SP) cores organized as 30 streaming multiprocessor (SMs) and has 4GB of memory for storing data used by the kernel functions. A 64-bit Ubuntu 10.04 Linux operating system was used to host the simulations. A sequential i.e. CPU only version of *snppgu-sim4* was created and compared to *snppgu-sim4*. We designate this CPU only simulator as *snpcpu-sim4*. *snpcpu-sim4* is identical to *snppgu-sim4* except for the computation of equation (2). Figure 4 shows the running times of the simulators with Π_1 as the SNP system.

The run times per SNP system are shown using three different time measurements: the *real* time, *user* time, and *sys* time taken using the Ubuntu Linux command *time*, based on the Unix command of the same name. The *real* time is the time that has elapsed during the run of the program (a ‘wall clock’ time measurement). The *user* time is the time spent by the program running in the CPU while in *user mode*. The *sys* time is the total CPU time used by the OS on behalf of the program that is being measured, and while the process is in *kernel mode*. A program or process in kernel mode means the process can use system calls or services such as allocating memory for itself, including hardware access (a more privileged

execution mode) while being in user mode means the program is usually restricted to its initial resources only (less privileged execution mode) [17].

In Figure 4 we see the large improvement of *snpgpu-sim4* over *snpcpu-sim4*, as expected. As expected also, *snpcpu-sim4* used up more time from the CPU as seen in the *real* and *sys* times. It's worth mentioning that *snpgpu-sim4* used a bit more of the CPU in the *user* times (though still significantly less than *snpcpu-sim4*) because *snpgpu-sim4* still needed some work from the CPU to process the inputs. Another noteworthy point is that with all three runtime figures (Figure 4 to 6) the *user* run time is significantly far less compared to the other two time measurements because it only measures the time used by the program alone in the CPU, and no other programs are involved in the time measurement. Table 1 summarizes averages of the kernel function runtimes and the CPU counter parts of the kernel functions as well as the average speedups. The maximum size, in terms of the number of neurons ($C_{k_{num}}$) and rules (R_{num}) of a system, that the current setup can simulate is given by $C_{k_{num}} = 4GBytes / (16Bytes + 4Bytes \times R_{num})$.

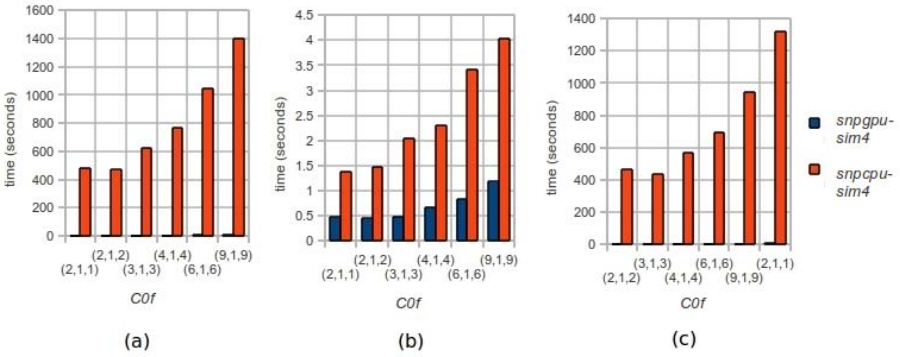


Fig. 4. Runtime graph of *snpgpu-sim4* versus *snpcpu-sim4* for Π_1 showing (a) *real*, (b) *user*, and (c) *sys* times usage

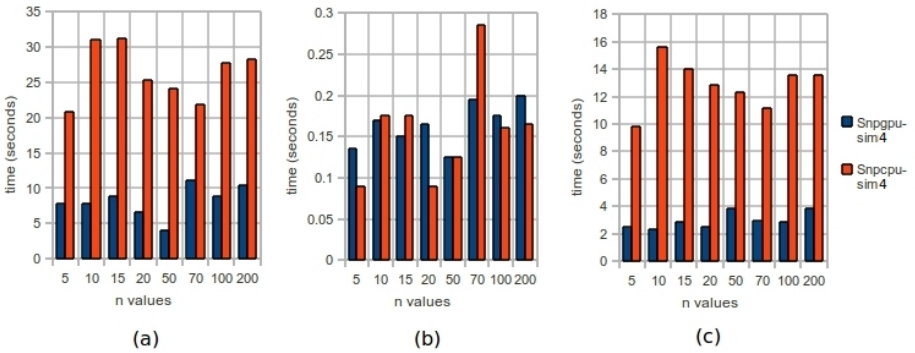


Fig. 5. Runtime graph of *snpgpu-sim4* versus *snpcpu-sim4* for Π_2 showing (a) *real*, (b) *user*, and (c) *sys* times usage

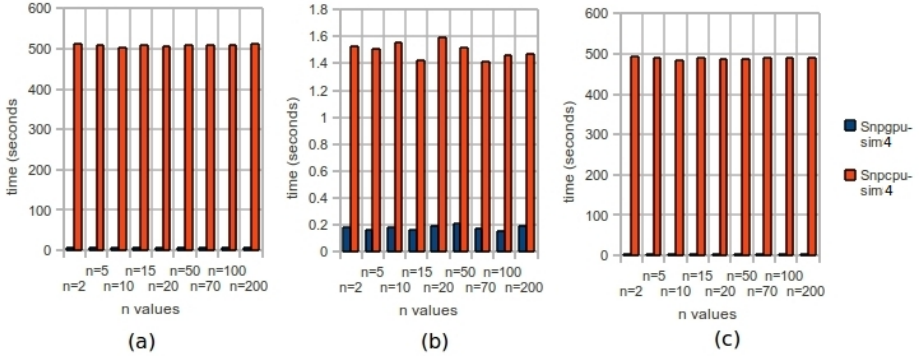


Fig. 6. Runtime graph of *snppgu-sim4* versus *snpcpu-sim4* for Π_3 showing (a) *real*, (b) *user*, and (c) *sys* times usage

Table 1. Summary of averages: kernel and CPU times, and speedup. All time measurements are in seconds, except for KRTA which is in microseconds. RTSA is Real Time Speedup Average, UTSA is User Time Speedup Average, STSA is System Time Speedup Average. KRTA is the Kernel Runtime Average, the amount of time the kernel function spent running inside the GPU/device. CRTA is the CPU Runtime Average, the amount of CPU time used by the CPU only (i.e. sequential) counterpart of the kernel function.

| | RTSA | UTSA | STSA | KRTA | CRTA |
|---------|----------------|--------------|----------------|-----------------------|-----------|
| Π_1 | 156.1439811343 | 3.5999180999 | 178.3754195194 | 107.33688871 μ s | 3.8535563 |
| Π_2 | 3.2014649226 | 0.9619771863 | 4.3513513514 | 216.442000587 μ s | 3.938559 |
| Π_3 | 67.0445847755 | 8.4018691589 | 192.8963174046 | 153.418998544 μ s | 3.9137748 |

Acknowledgments. Francis Cabarle is supported by the DOST-ERDT program. Henry Adorna is funded by the DOST-ERDT research grant and the Alexan professorial chair of the UP Diliman Department of Computer Science. M.A. Martínez-del-Amor and M.J. Pérez-Jiménez are supported by “Proyecto de Excelencia con Investigador de Reconocida Valía” of the “Junta de Andalucía” under grant P08-TIC04200, and by the project TIN2009-13192 of the “Ministerio de Educación y Ciencia” of Spain, both co-financed by FEDER funds.

References

1. Cabarle, F., Adorna, H., Martínez-del-Amor, M.A.: An Improved GPU Simulator For Spiking Neural P Systems. Accepted in the IEEE Sixth International Conference on Bio-Inspired Computing: Theories and Applications, Penang, Malaysia (September 2011)
2. Cabarle, F., Adorna, H., Martínez-del-Amor, M.A.: A Spiking Neural P system simulator based on CUDA. Accepted in the Twelfth International Conference on Membrane Computing, Paris, France (August 2011)

3. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs. *Journal of Logic and Algebraic Programming* 79(6), 317–325 (2010)
4. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics* 11(3), 313–322 (2010)
5. Chen, H., Ionescu, M., Ishdorj, T.-O., Păun, A., Păun, G., Pérez-Jiménez, M.: Spiking neural P systems with extended rules: universality and languages. *Natural Computing: an International Journal* 7(2), 147–166 (2008)
6. Ciobanu, G., Wenyuan, G.: P Systems Running on a Cluster of Computers. In: Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *WMC 2003*. LNCS, vol. 2933, pp. 123–139. Springer, Heidelberg (2004)
7. Díaz, D., Graciani, C., Gutiérrez, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Software for P systems. In: Păun, G., Rozenberg, G., Salomaa, A. (eds.) *The Oxford Handbook of Membrane Computing*, ch. 17, pp. 437–454. Oxford University Press, Oxford (2009)
8. Fatahalian, K., Sugeran, J., Hanrahan, P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS 2004)*, pp. 133–137. ACM, NY (2004)
9. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. *Communications of the ACM* 53(11), 58–66 (2010)
10. Harris, M.: Mapping computational concepts to GPUs. In: *ACM SIGGRAPH 2005 Courses*, NY, USA (2005)
11. Ionescu, M., Păun, G., Yokomori, T.: Spiking Neural P Systems. *Journal Fundamenta Informaticae* 71(2,3), 279–308 (2006)
12. Kirk, D., Hwu, W.: *Programming Massively Parallel Processors: A Hands On Approach*, 1st edn. Morgan Kaufmann, MA (2010)
13. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: *PyCUDA: GPU Run-Time Code Generation for High-Performance Computing*. Scientific Computing Group, Brown University, RI, USA (2009)
14. Nguyen, V., Kearney, D., Gioiosa, G.: A Region-Oriented Hardware Implementation for Membrane Computing Applications and Its Integration into Reconfig-P. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) *WMC 2009*. LNCS, vol. 5957, pp. 385–409. Springer, Heidelberg (2010)
15. NVIDIA corporation, *NVIDIA CUDA C programming guide*, version 3.0. NVIDIA, CA, USA (2010)
16. Păun, G., Ciobanu, G., Pérez-Jiménez, M. (eds.): *Applications of Membrane Computing*. Natural Computing Series. Springer, Heidelberg (2006)
17. Stallings, W.: *Operating systems: internals and design principles*, 6th edn. Pearson/Prentice Hall, NJ, USA (2009)
18. Zeng, X., Adorna, H., Martínez-del-Amor, M.A., Pan, L., Pérez-Jiménez, M.: Matrix Representation of Spiking Neural P Systems. In: Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *CMC 2010*. LNCS, vol. 6501, pp. 377–391. Springer, Heidelberg (2010)